

# Dirty Tricks for Moar FPS on Tilers

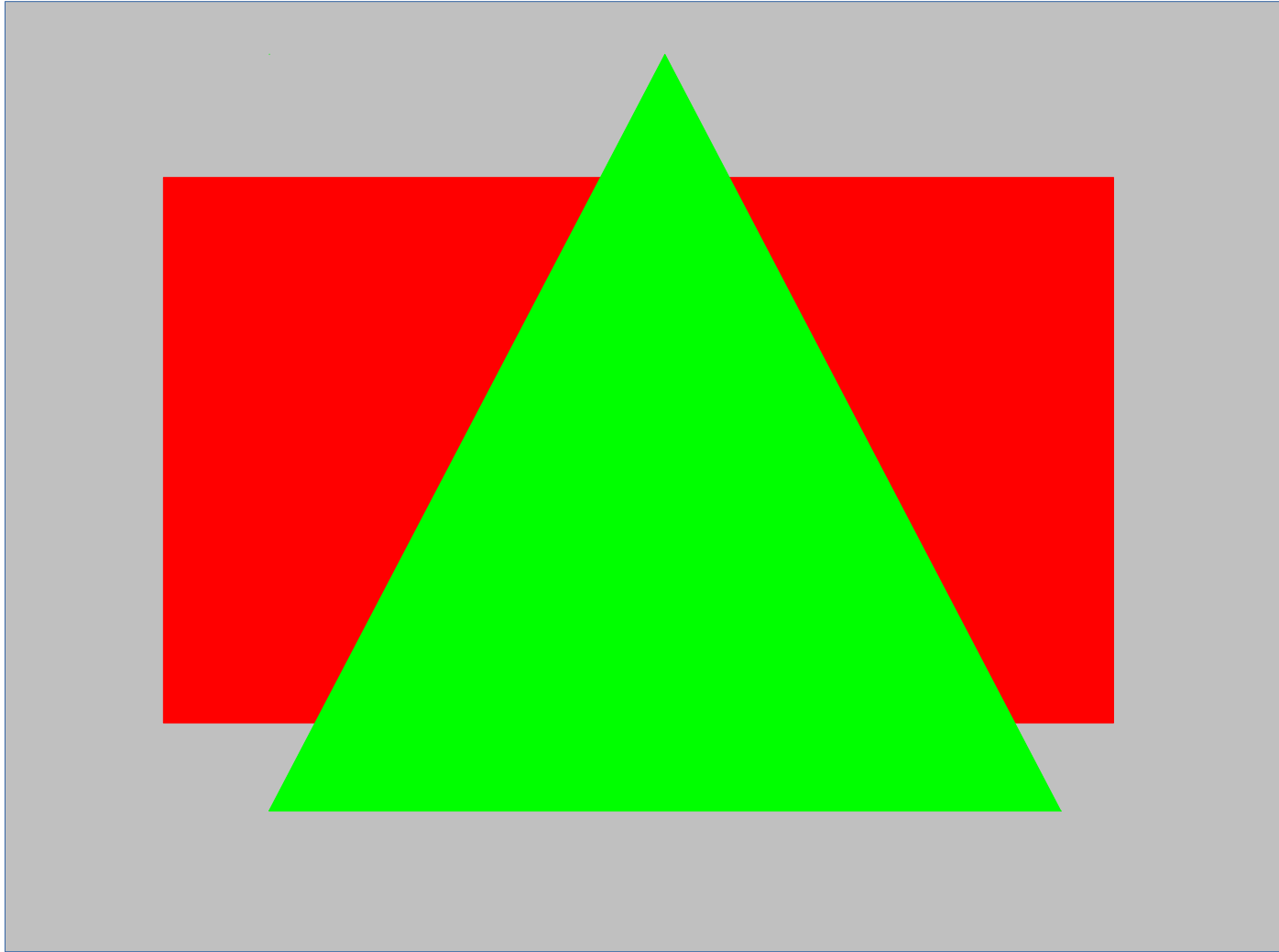
Batch Reordering and Resource Shadowing

Rob Clark  
Red Hat  
(freedreno)

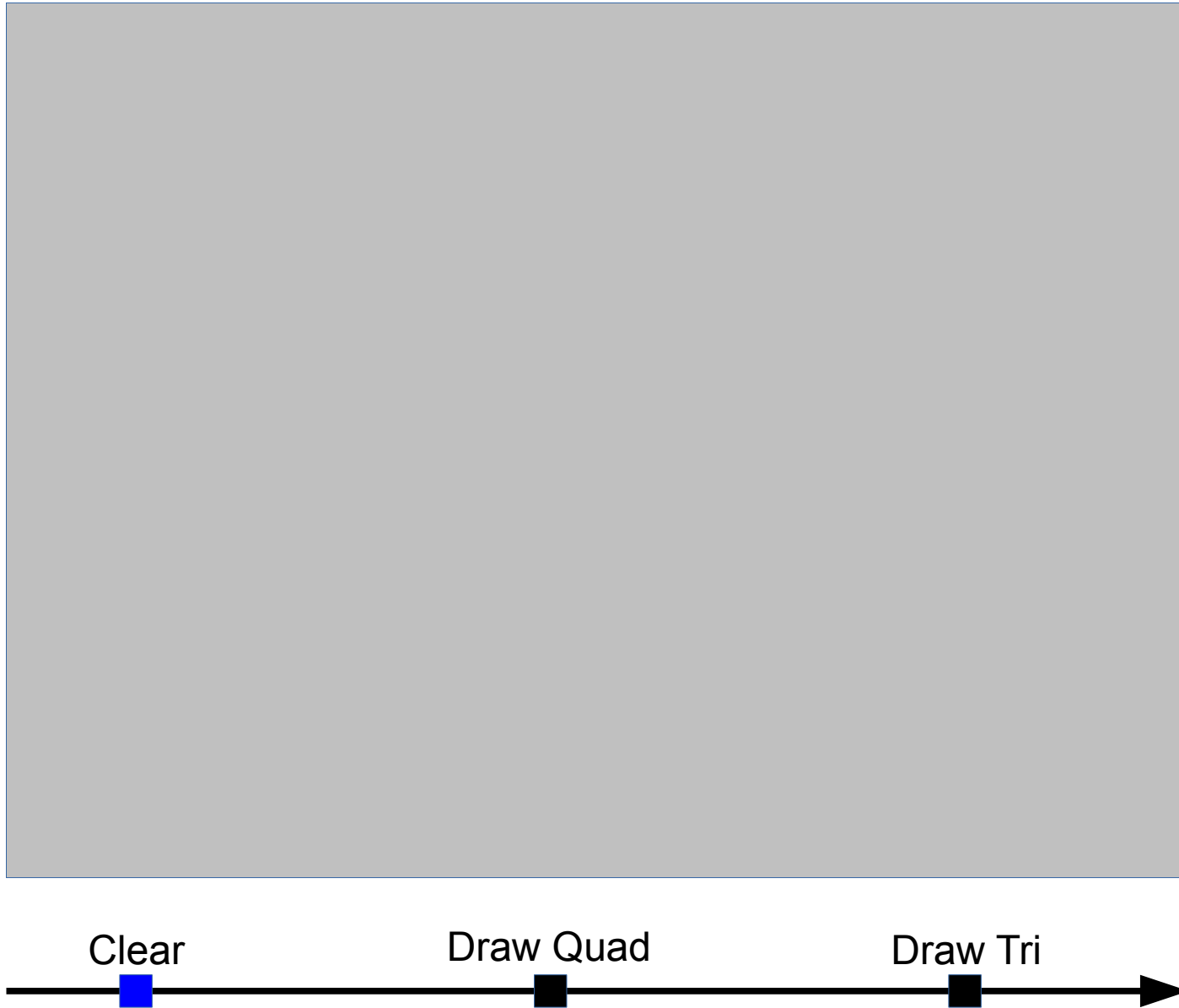
# Problem Statement

- Tiler GPU's optimize/reduce memory bandwidth requirements by rendering per-tile with mrt/color and depth/stencil in small internal tile buffer
- But many anti-patterns exist in GL programs that cause unnecessary flush/restore
  - Unnecessary FBO switches
  - Mid-frame texture uploads or UBO updates
- With some driver cleverness we can reduce this
  - Batch reordering (aka job reshuffling)
  - Resource shadowing (aka ghosting)

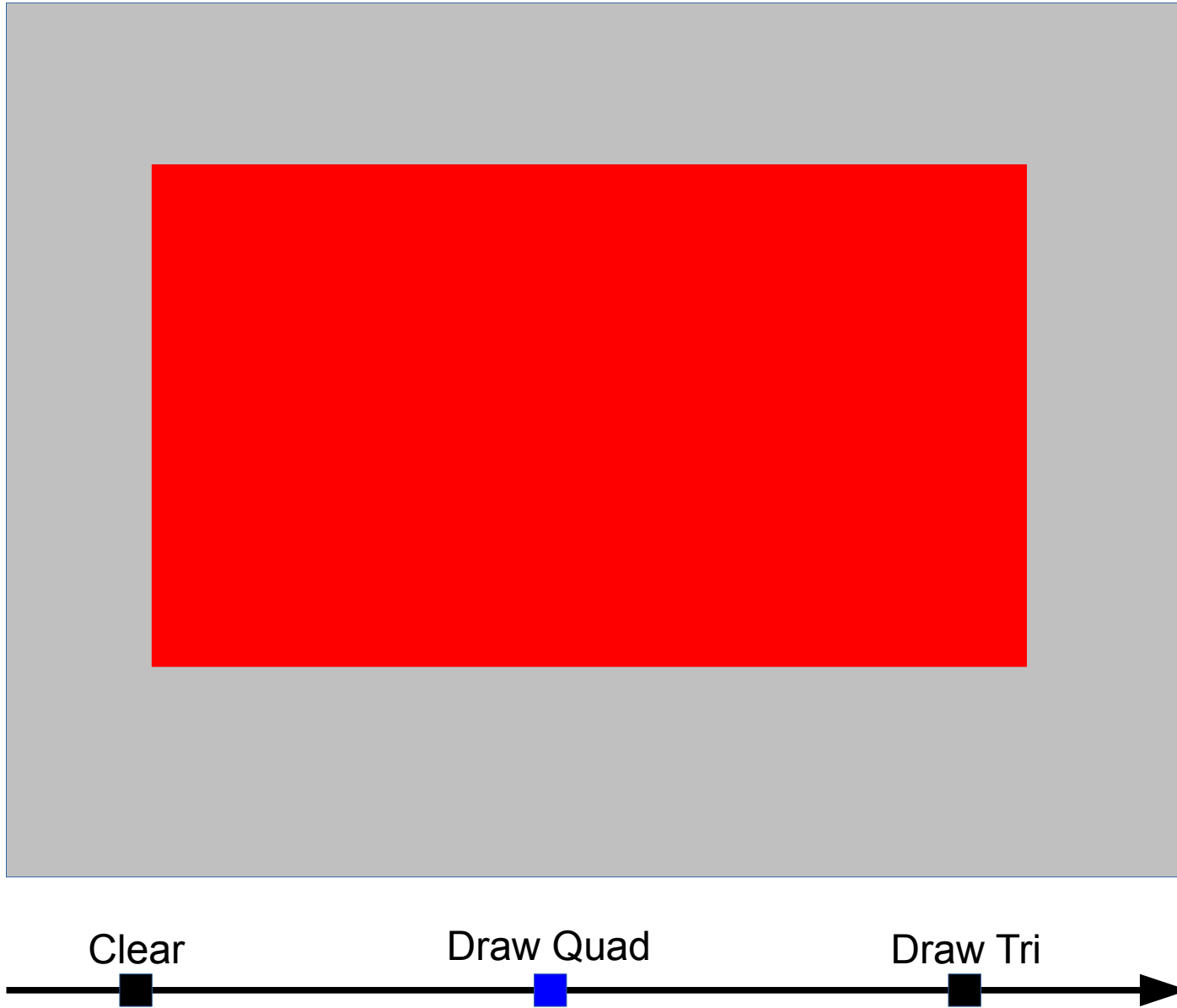
# Example super-awesome FPS game: triangle-quad



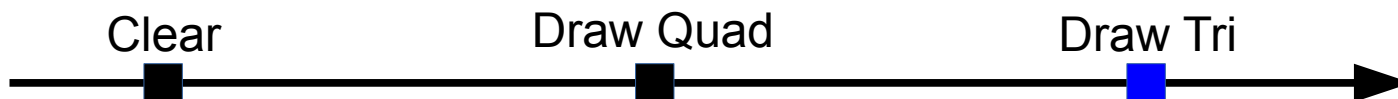
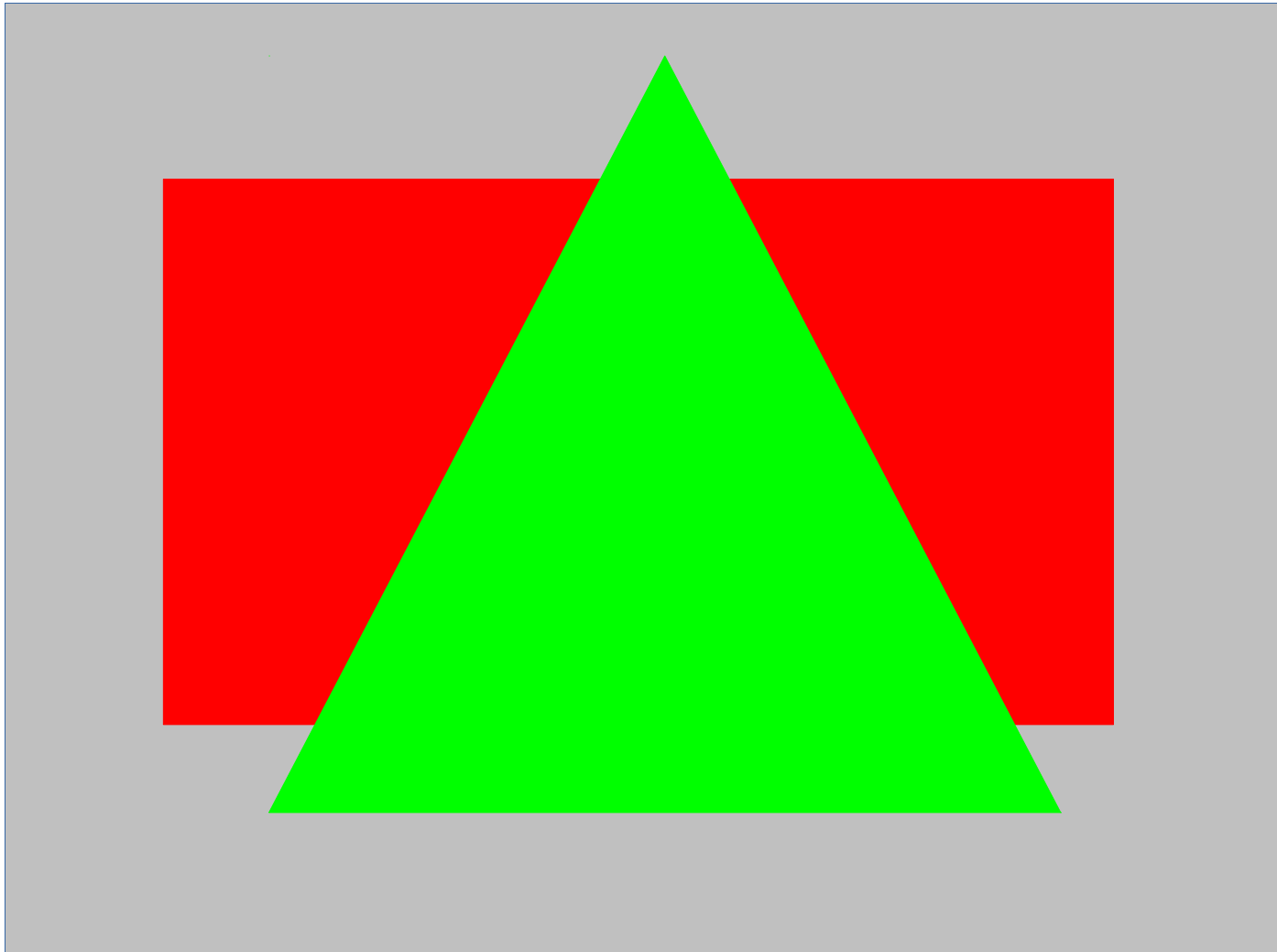
# Traditional GPU:



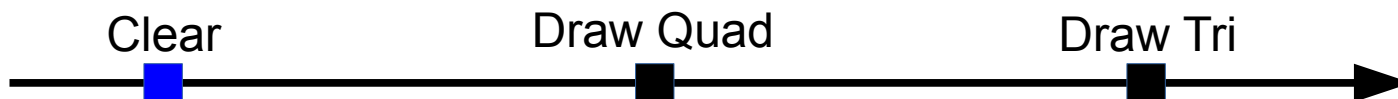
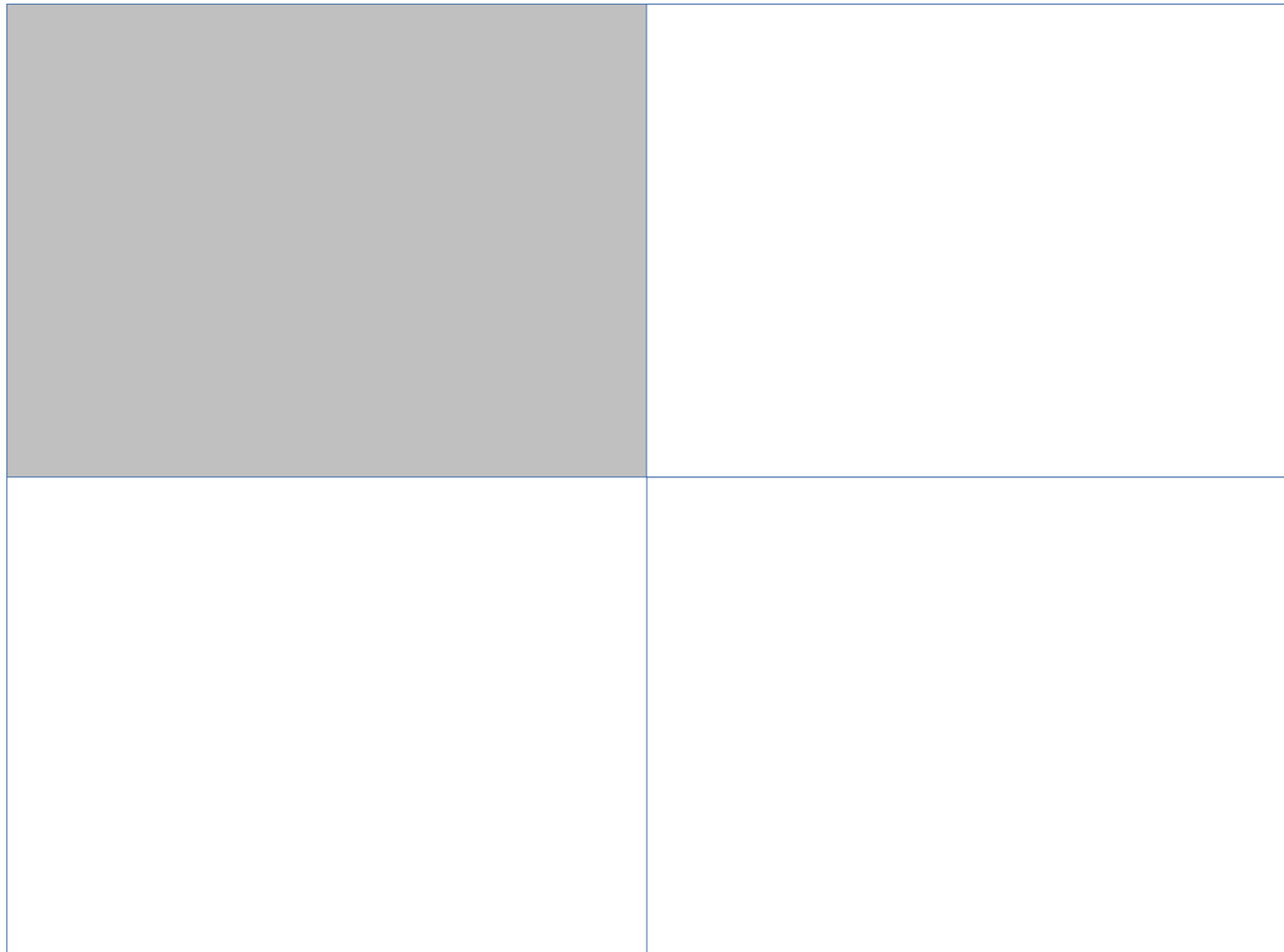
# Traditional GPU:



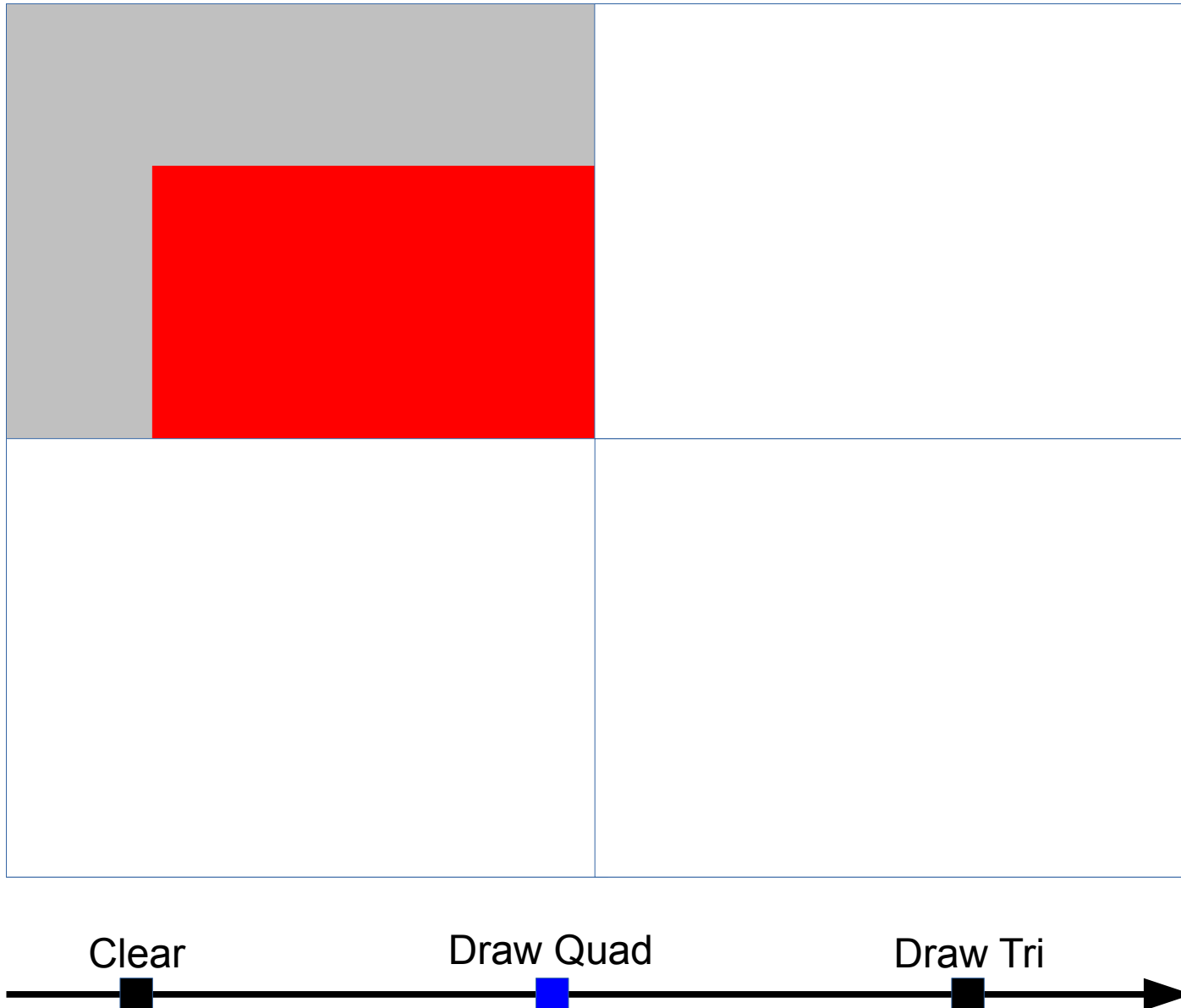
# Traditional GPU:



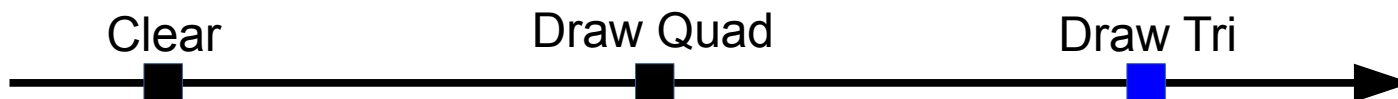
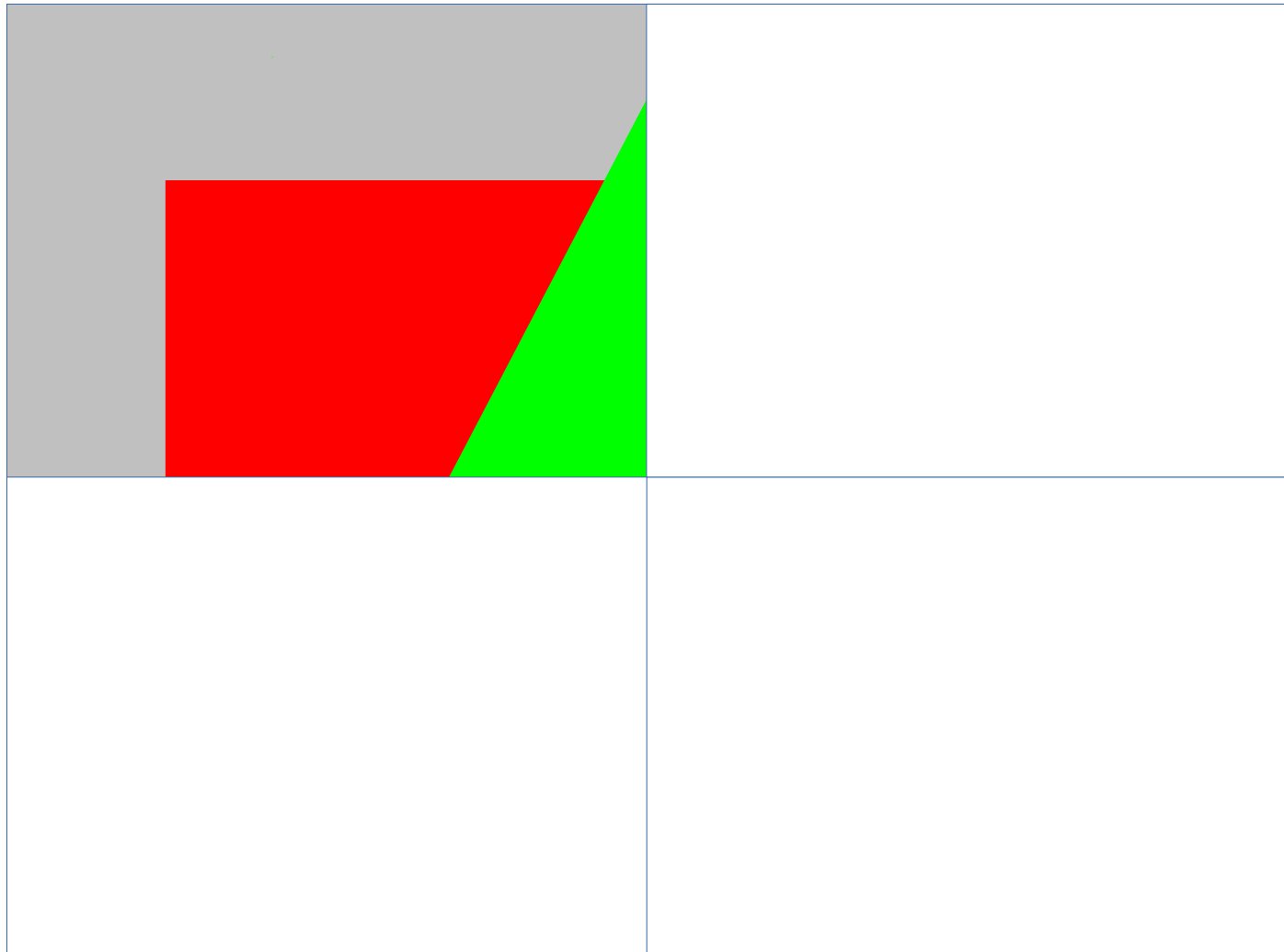
# Tiler GPU:



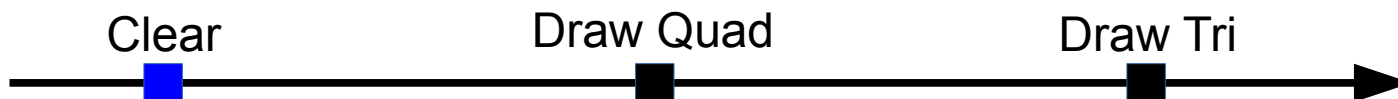
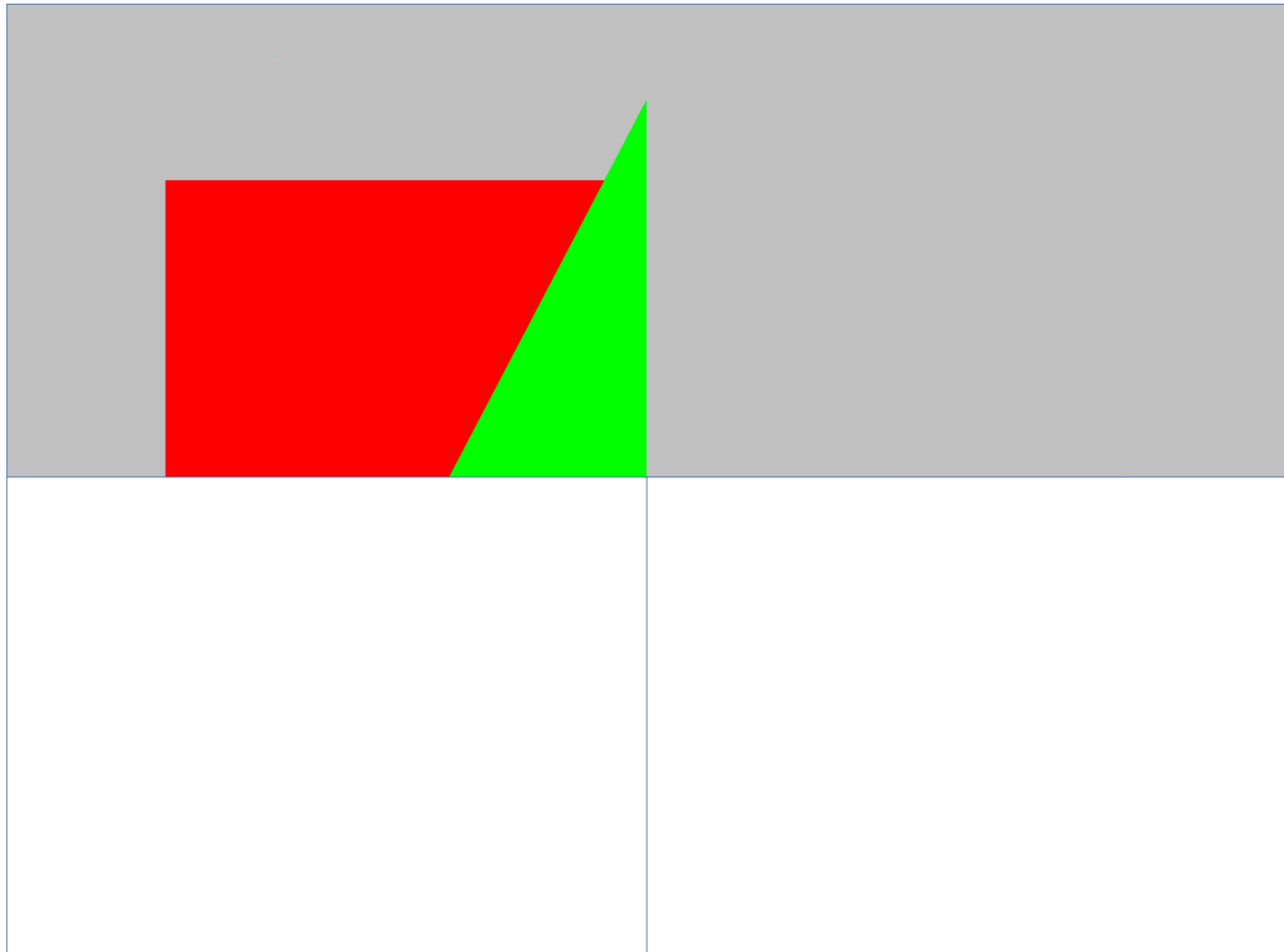
# Tiler GPU:



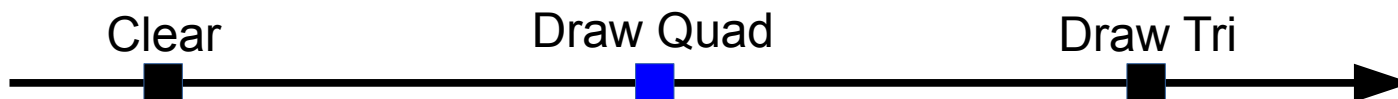
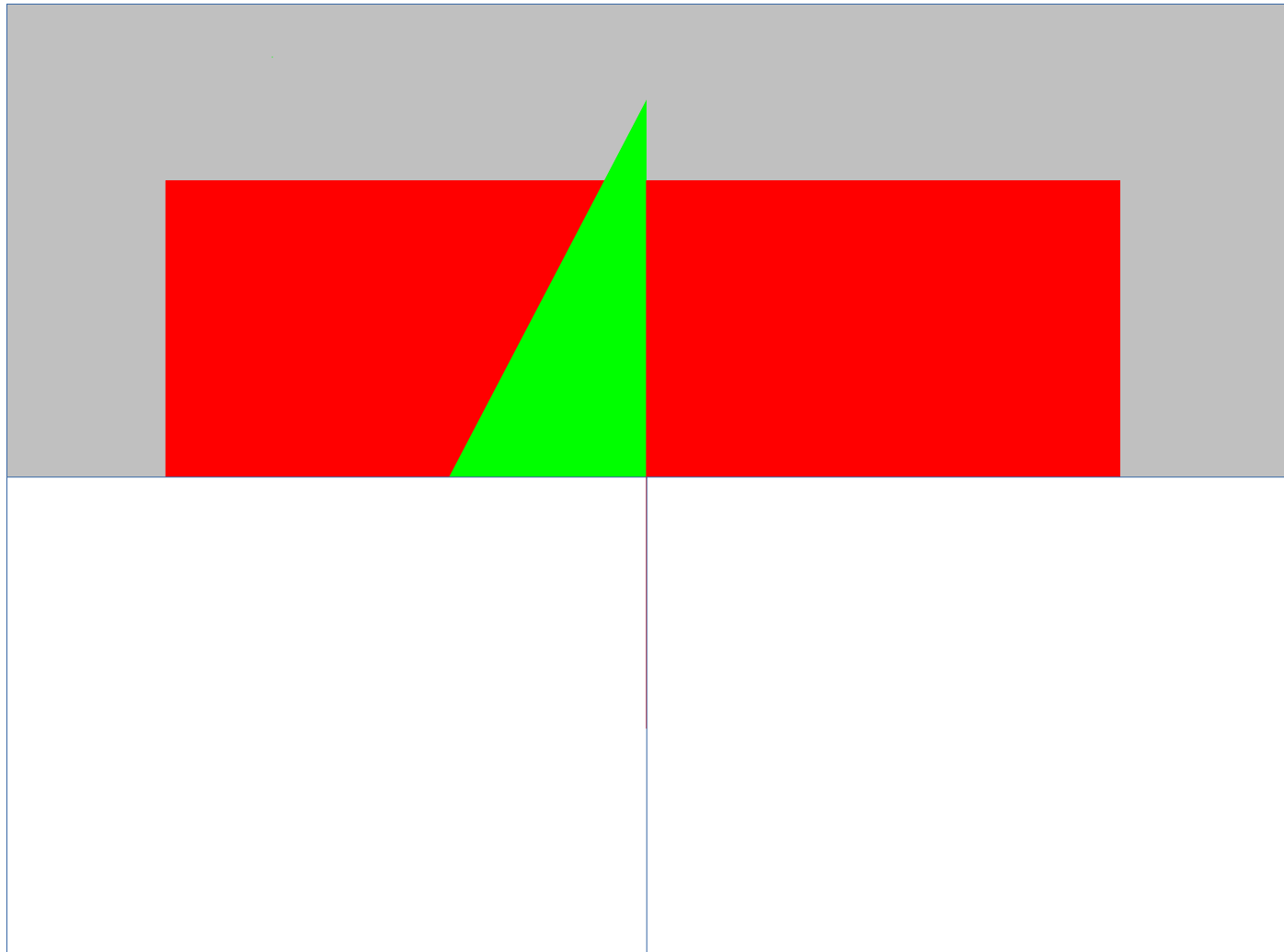
# Tiler GPU:



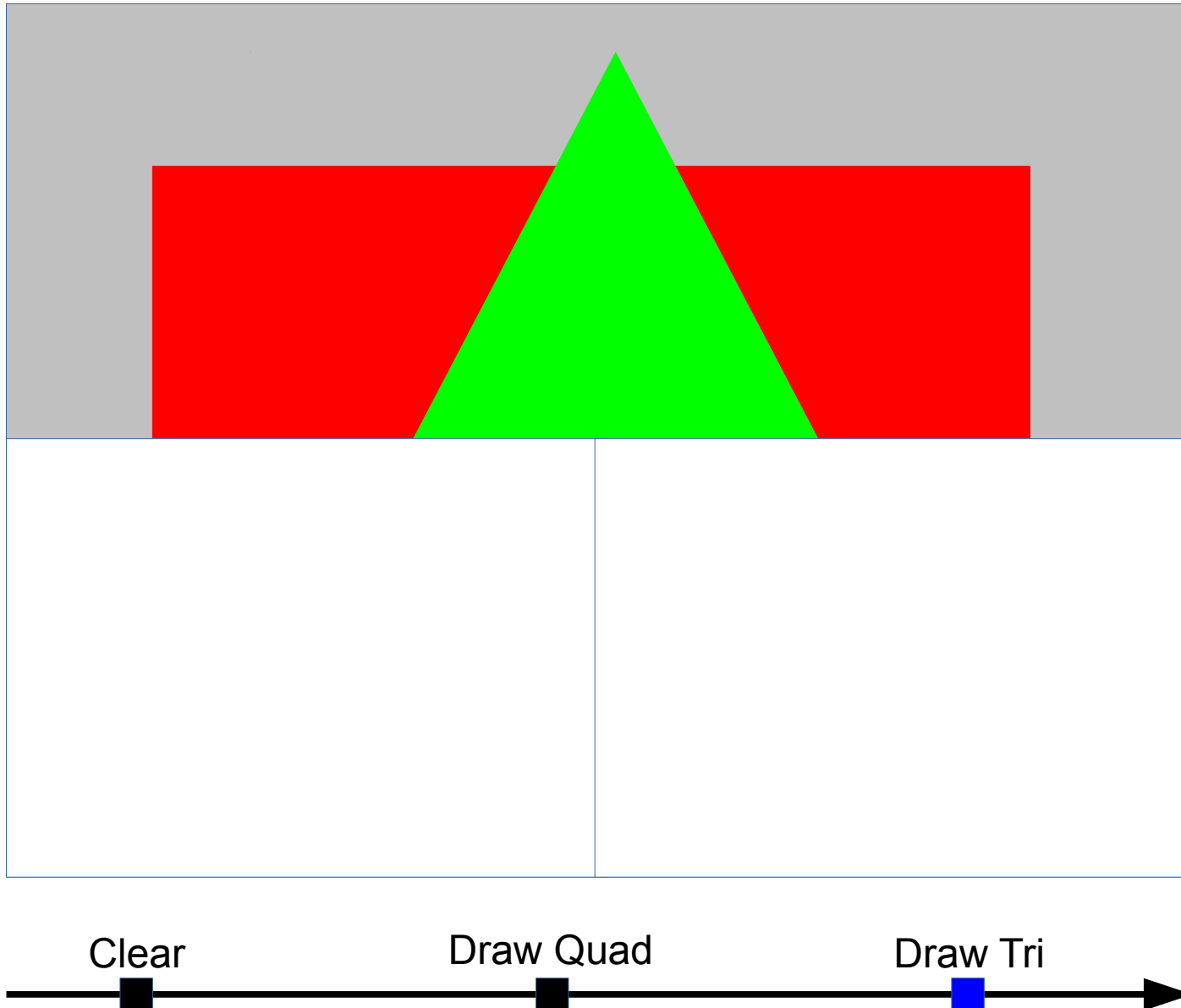
# Tiler GPU:



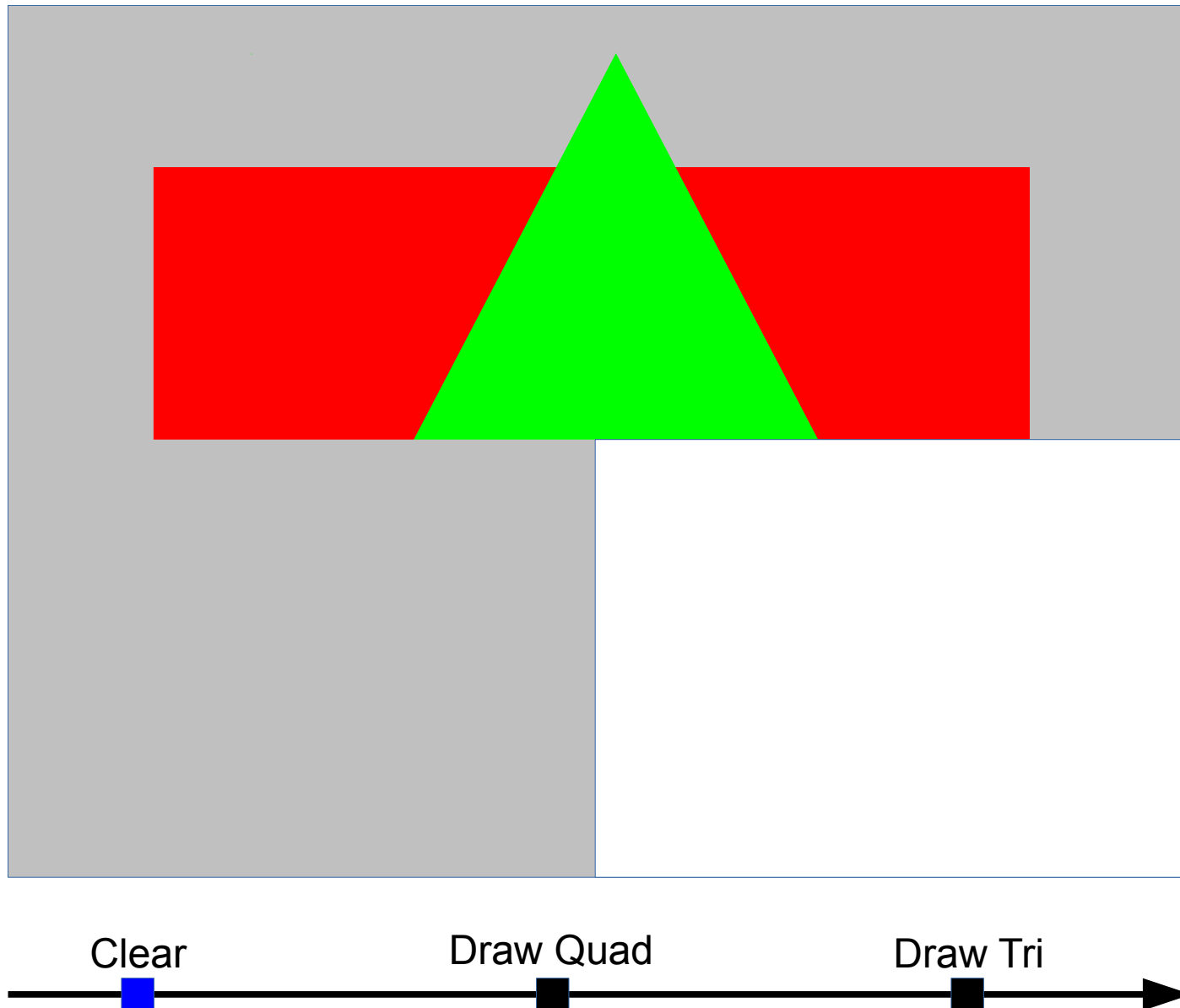
# Tiler GPU:



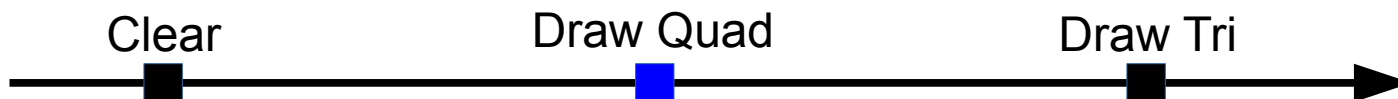
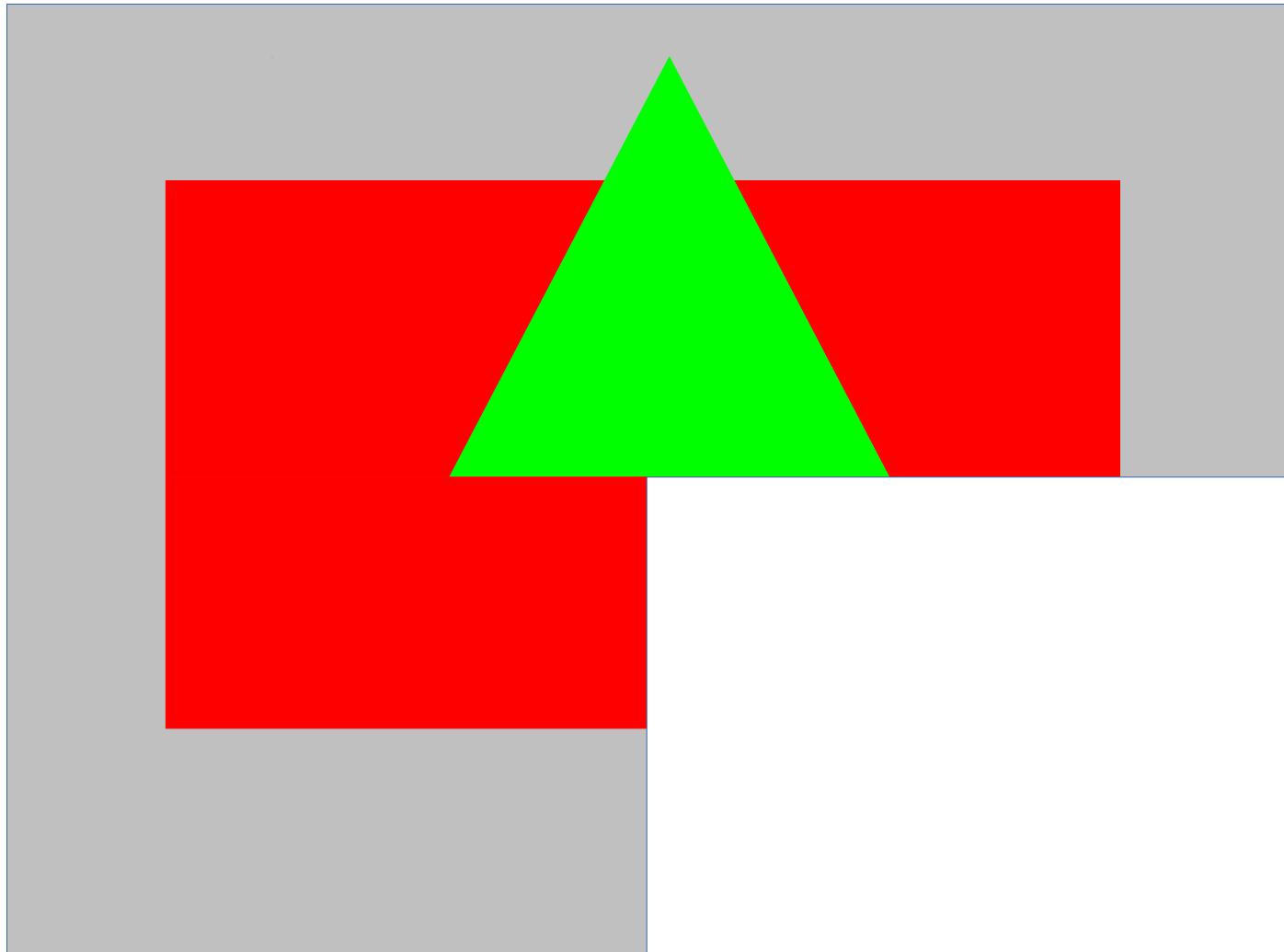
# Tiler GPU:



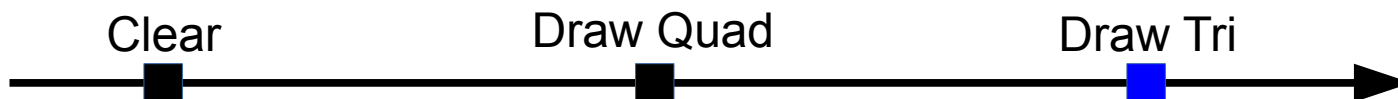
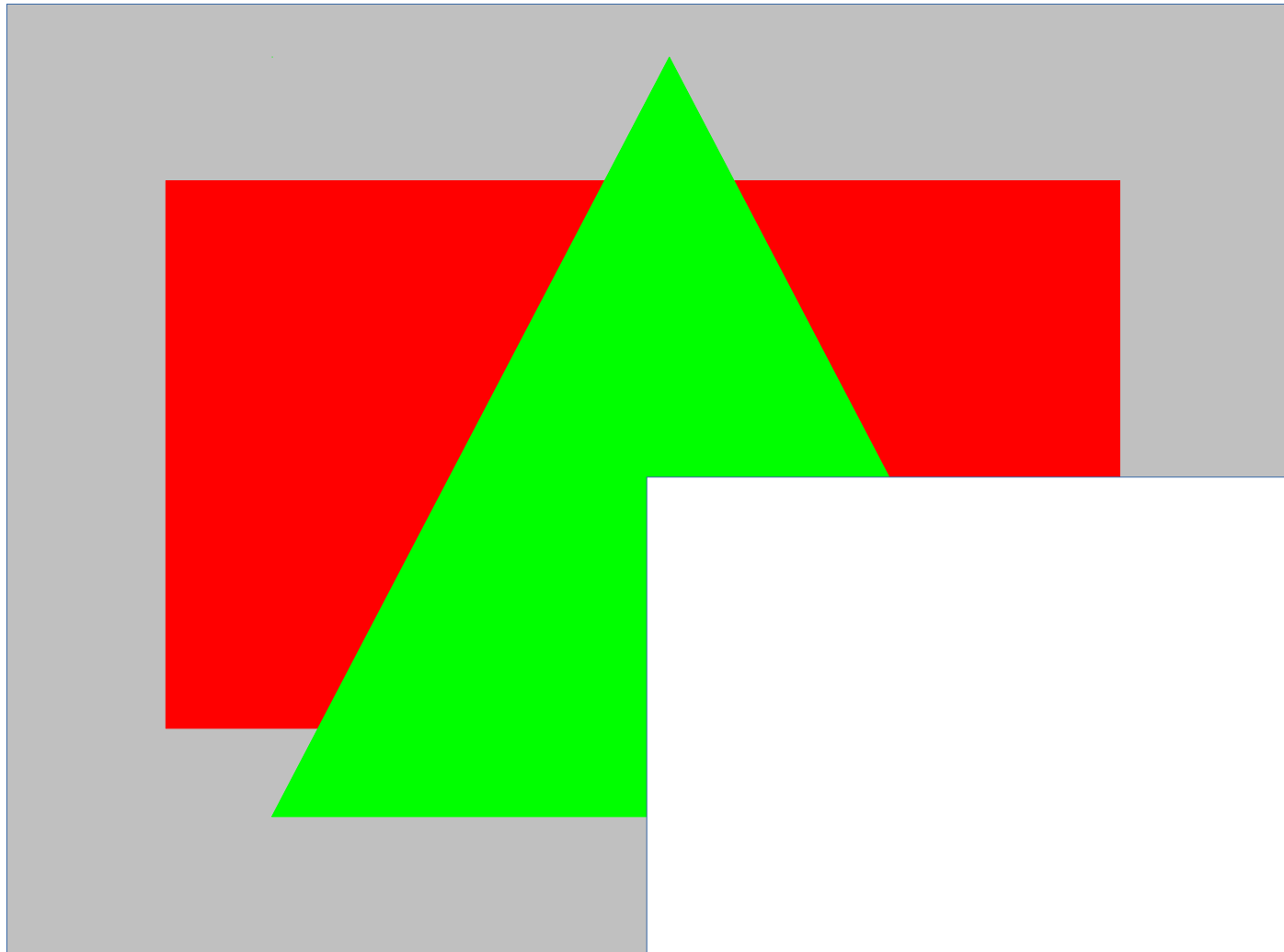
# Tiler GPU:



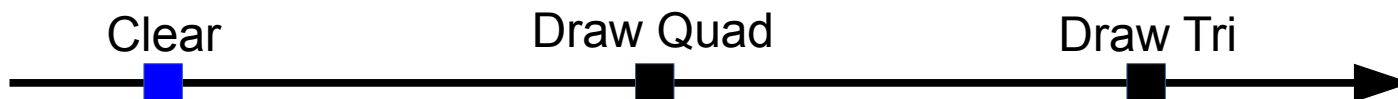
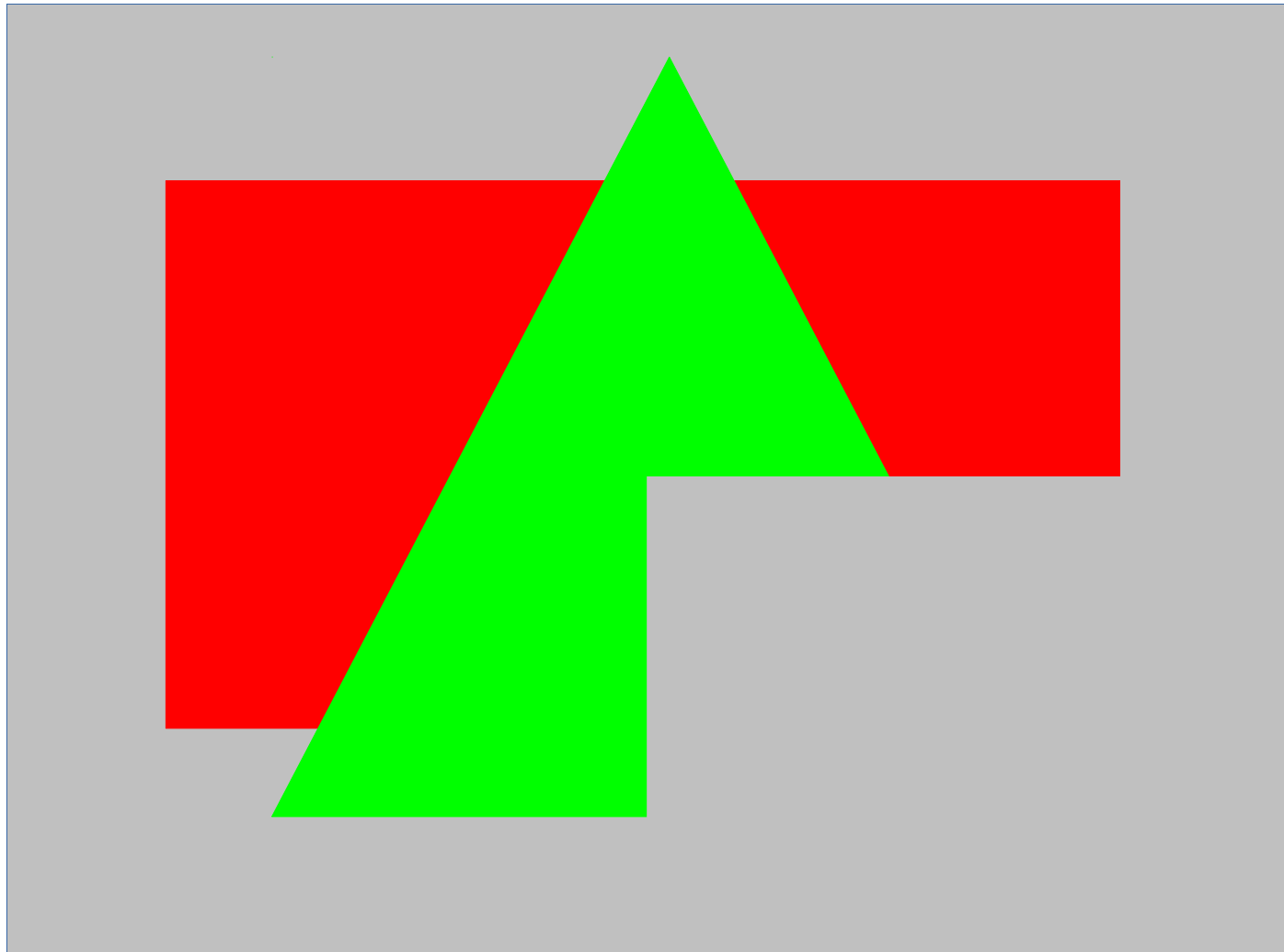
# Tiler GPU:



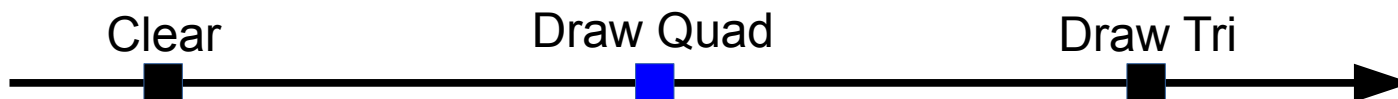
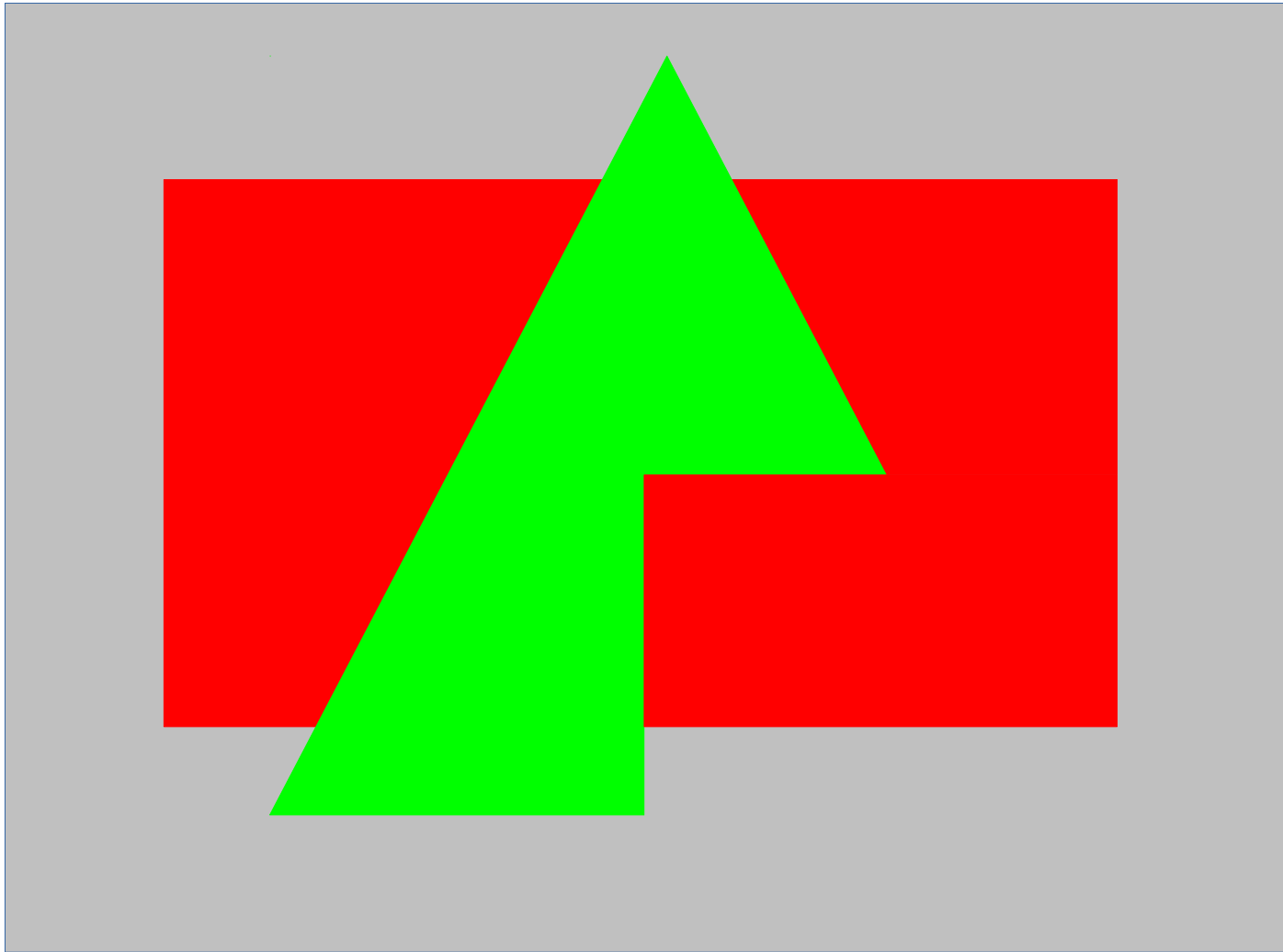
# Tiler GPU:



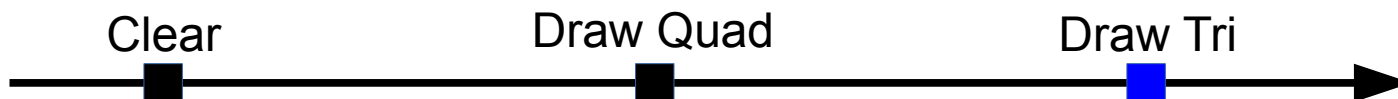
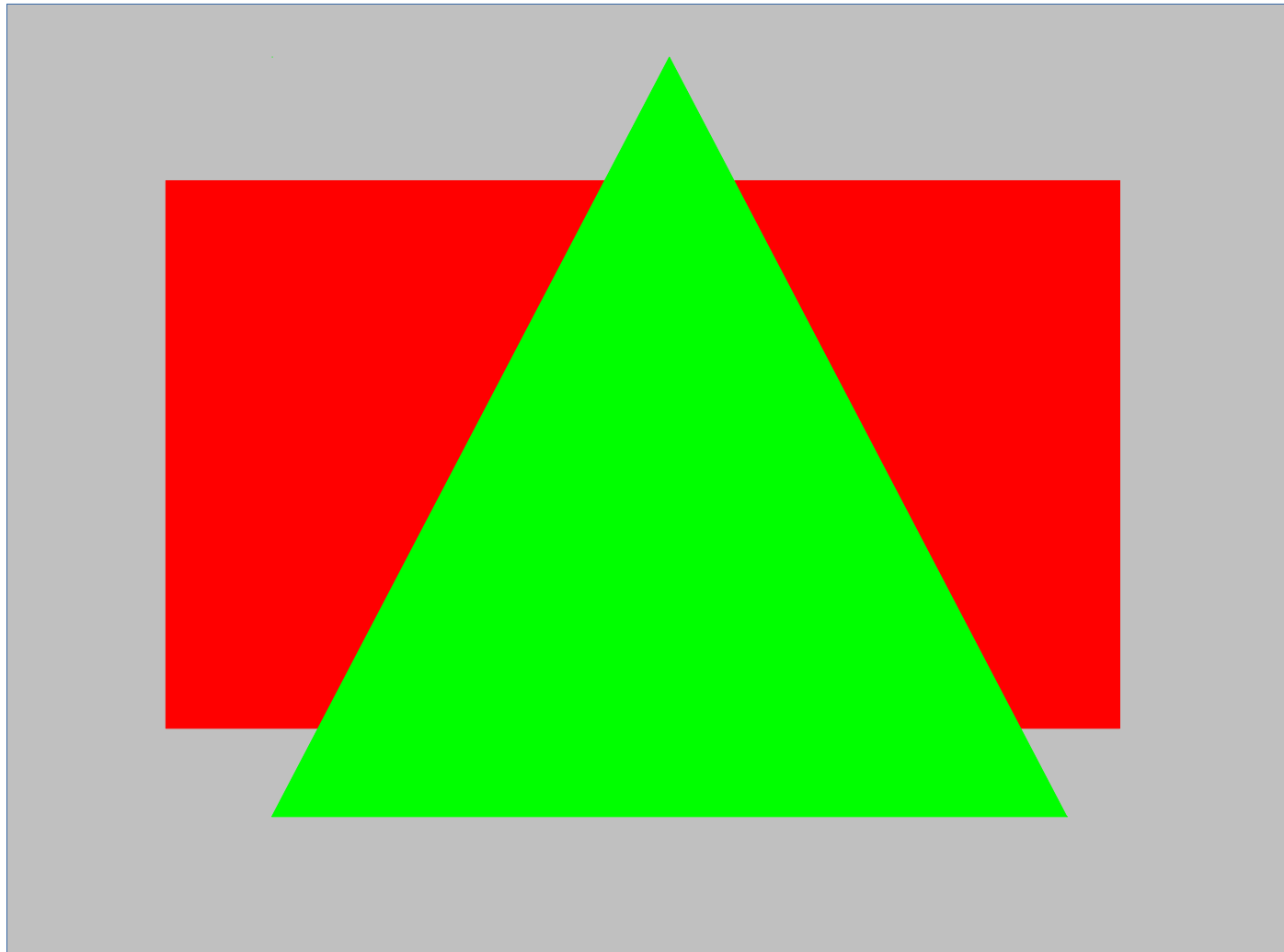
# Tiler GPU:



# Tiler GPU:



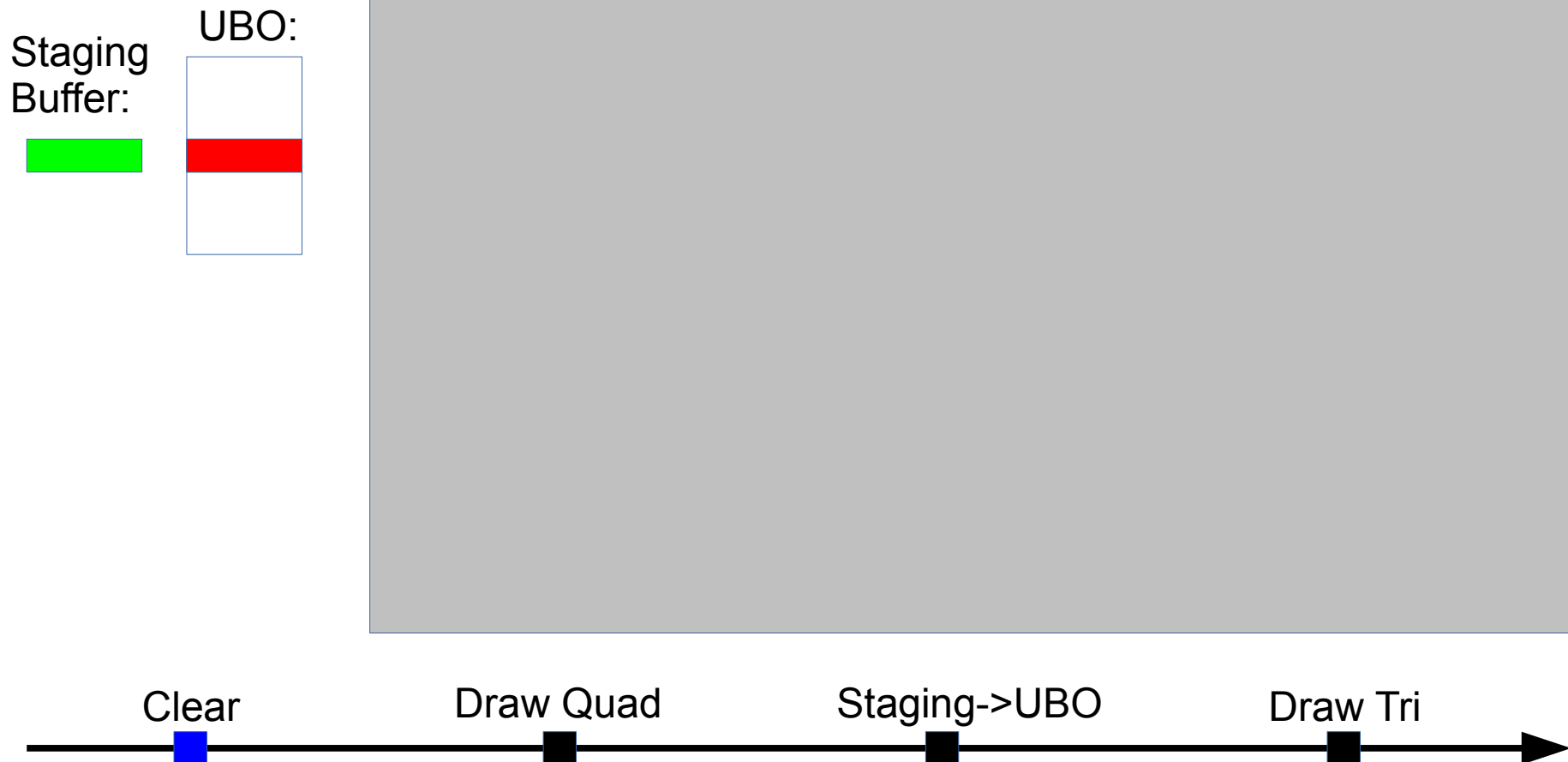
# Tiler GPU:



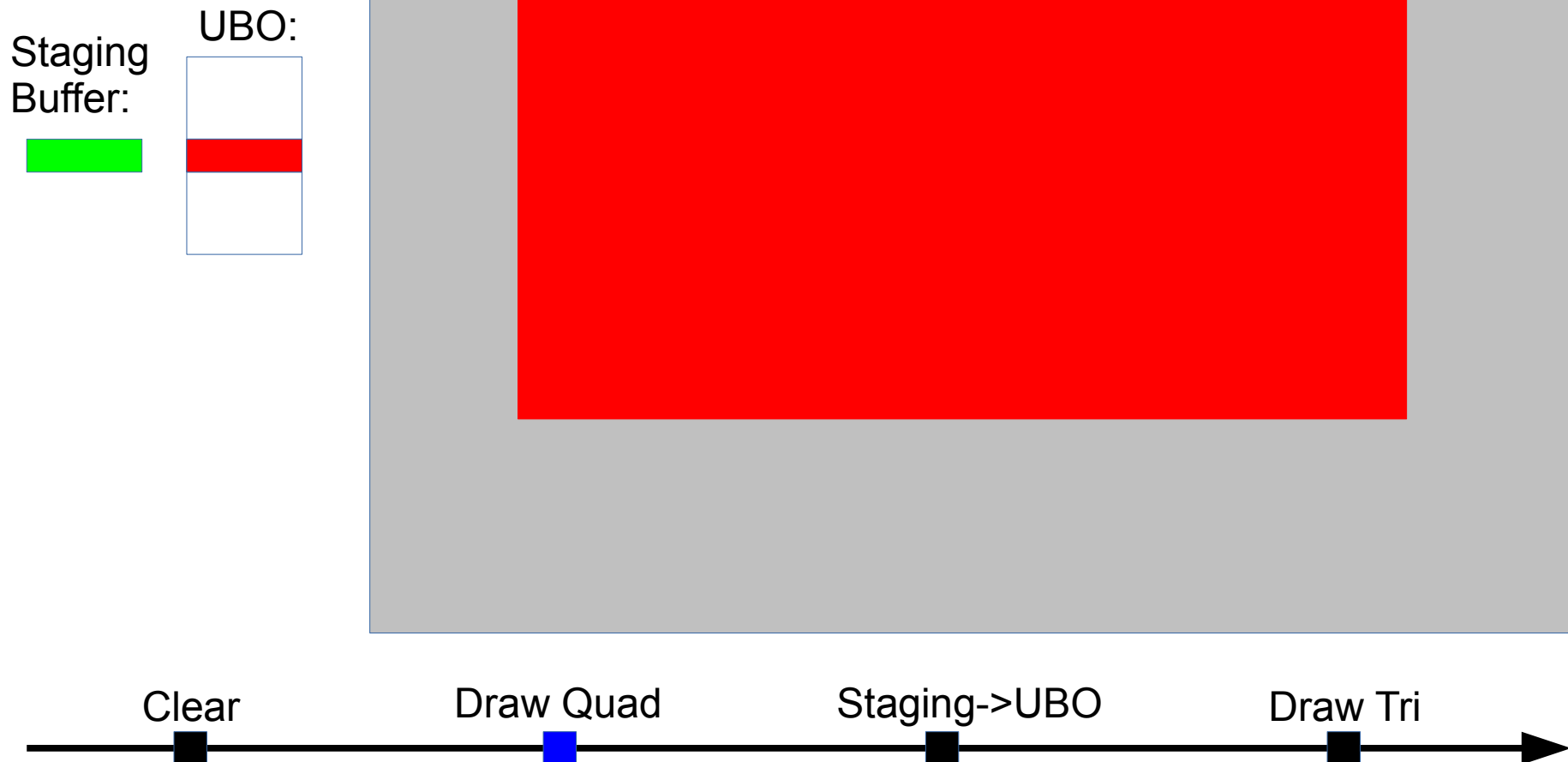
# But...

- This is a super-modern game using a UBO to pass color to FS
  - Mid-frame UBO update to change color
- Similar scenario for mid-frame texture uploads
  - but this was an easier example to draw
- Typically a non-tiler GPU driver would use a staging buffer to upload new data to modified buffer

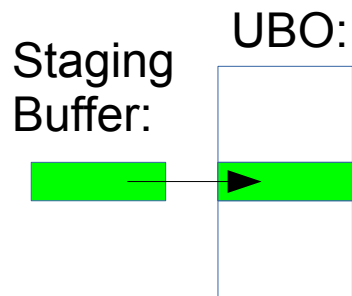
# Traditional GPU:



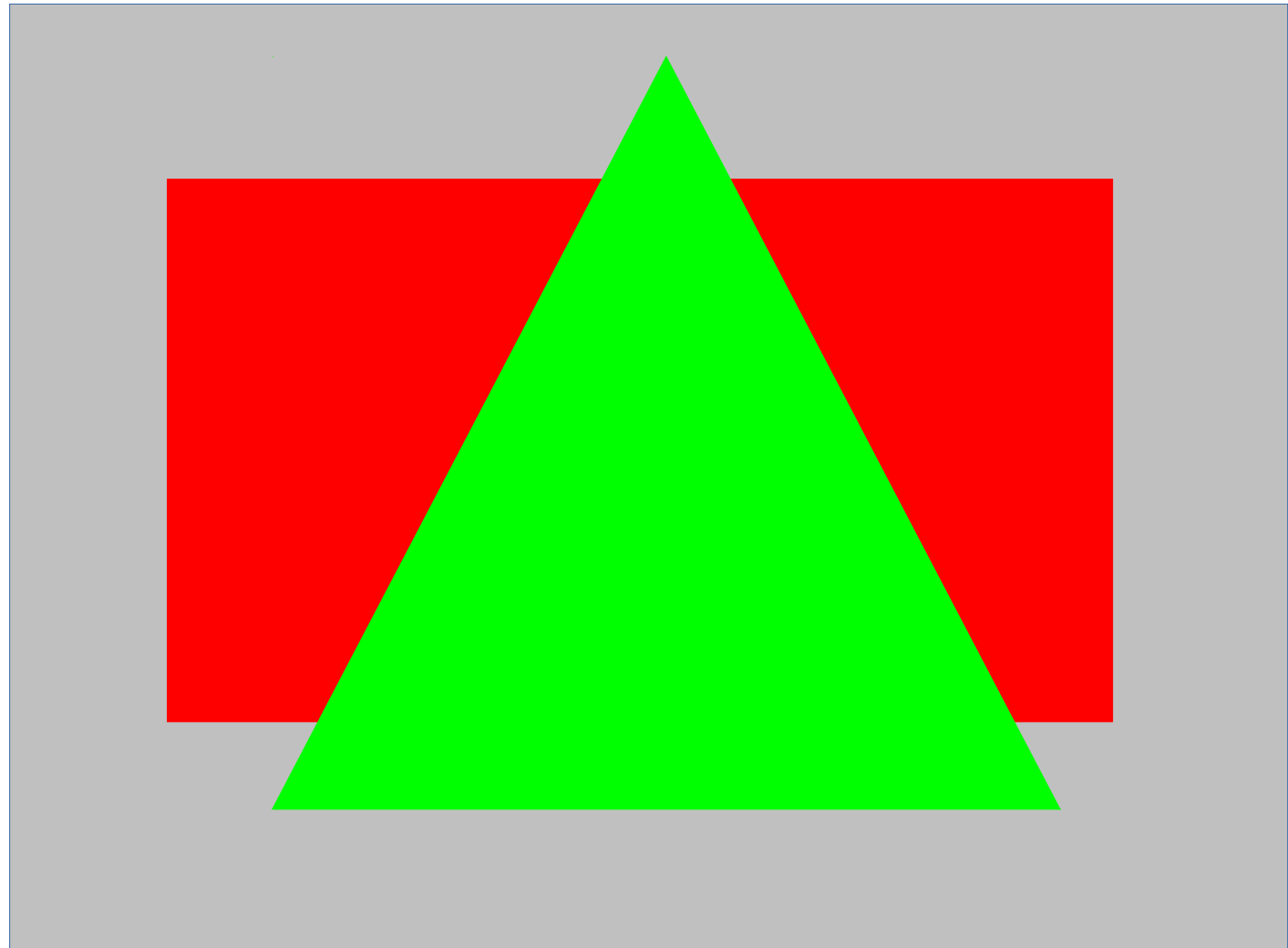
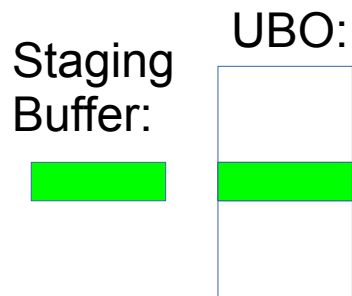
# Traditional GPU:



# Traditional GPU:



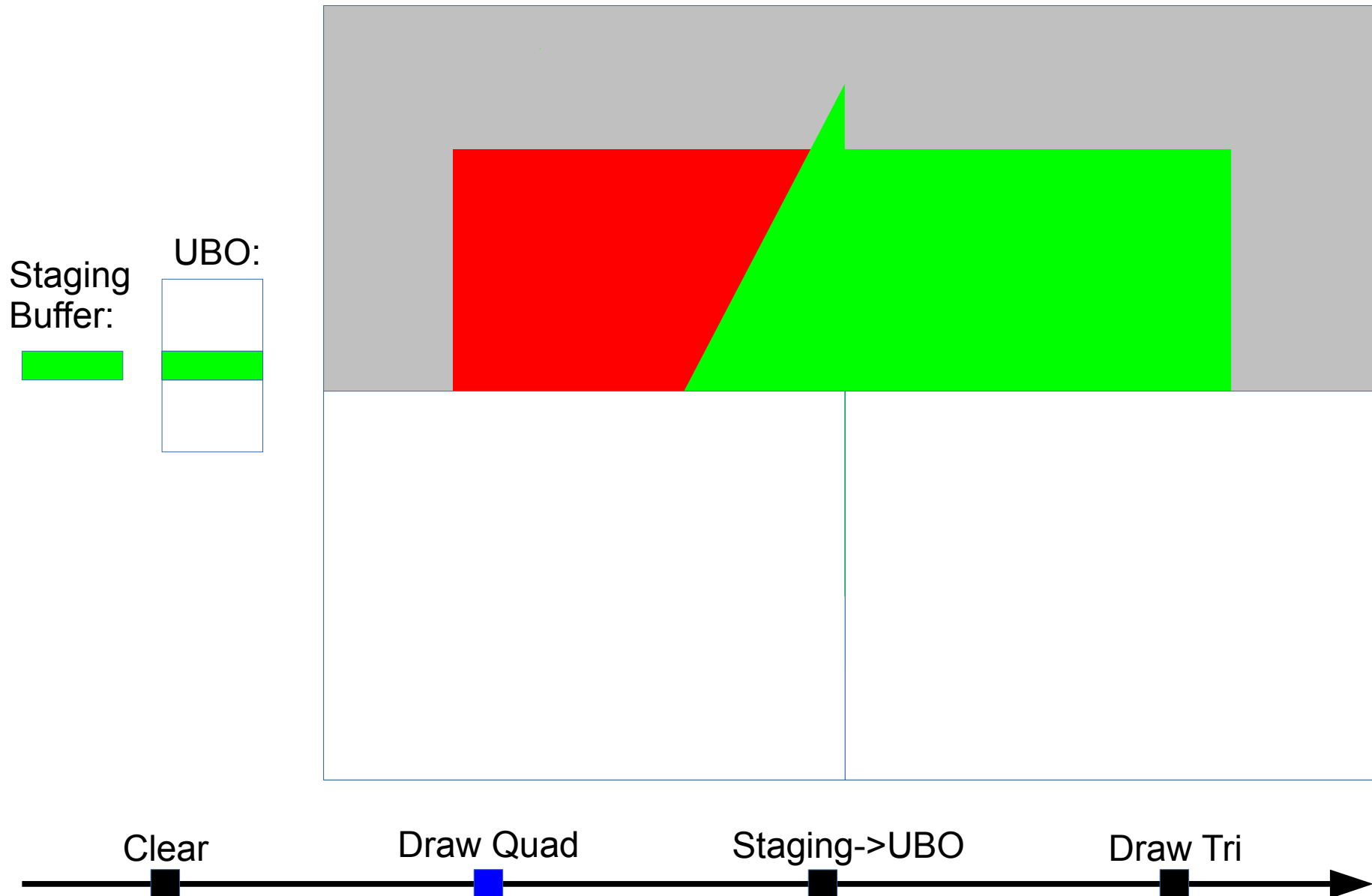
# Traditional GPU:



# But...

This doesn't work so well for a tiled gpu

# Tiler GPU:



# Naive/Previous Solution...

- Flush on mid-frame resource (UBO/texture/etc) update
- But this is expensive
  - RGBA8 @1080p => 8MB
  - z24s8 @ 1080p => 8MB
  - MRT and/or higher bpp formats (float16/float32) formats increase this proportionally
- Each unnecessary flush has a corresponding restore
  - To move data back into tile buffer..
  - So simple RGBA8 + z24s8 => each extra flush costs 16MB write bandwidth for flush, and 16MB read bandwidth for restore
  - With MRT (multiple render targets) and/or “exotic” formats this goes up

# So... to the dirty tricks

- We need to shadow resources
  - Buffers: UBOs, textures, etc
- Re-order rendering in case of FBO switches
  - This includes internally generated u\_blitter stuff like resource shadowing back-blits and mipmap level generation
- These two tricks are related
  - We don't have a separate dma pipe for blits / mipmap generation / etc
  - u\_blitter → everything looks like FBO switch!
- Fortunately, solving it this way handles FBO switches too
  - vs. special casing blits

# But how to implement? (1)

- Split out “batch” object
  - vc4 calls this a “job”
    - Basically a “tile pass”
  - Tracks command-stream and all state related to gmem/tile pass
    - Which render target buffers (mrt & z/s) are cleared
    - Stats which we use to decide about tiling/gmem vs bypass
    - Accumulated scissor (lets us skip many tiles for UI type workloads)
    - Patch-lists
    - Query result bo's
  - Some tiler gpu's handle this more automatically
    - But adreno requires the driver to handling the tiling in the driver via explicit cmdstream to handle restore and resolve
    - So all this state must move from context → batch so that it is still around / valid later when we flush and construct gmem/tiling cmdstream

# But how to implement? (2)

- Batch Cache
  - Construct a hash table key from `pipe_framebuffer_state`
  - Can't use pfb as-is because transient `pipe_surface` ptrs
- On FBO switch (`ctx→set_framebuffer_state()`)
  - Hashtable lookup to find exsisting unflushed batch
  - Otherwise create new batch and add to hash table

# But how to implement? (3)

- A bunch of dependency tracking
  - We need to track per resource:
    - N batches that read a resource
    - 1 batch that writes a resource
  - Per draw, look at dependencies of read and written resources
    - Textures, UBOs, VBOs, TF stream-out buffers, query result buffers, etc
    - Resources written by draw → dependency on other batches that read or write
    - Resources read by draw → dependency on batches that write
- Need to ensure batches are executed in correct order
  - ie. the batch that writes a resource must run before the one that reads it
    - For example batch that writes TF streamout buffer must run before batch that uses it as VBO
    - Or batch that writes MRT buffer must run before batch that uses it as texture
  - And the batch that overwrites a resource must run after any that read the previous version
  - So, per batch, track the N dependent batches
- Also needed to ensure the correct batches are flushed before a `transfer_map(READ)` or `transfer_map(WRITE)`

# First try..

- Track per pipe\_resource
  - last\_read\_batch
  - write\_batch
- Track single dependent batch per batch
- Low overhead (avoids hash set per bo)
- But introduces too many artificial dependencies

# Solving dependency tracking properly..

- Per batch
  - hash set of dependent batches
  - hash set of used (read/write) resources
- Per resource
  - hash set of batches that read the resource
  - single batch that writes the resource
- Hash sets are  $O(1)$  but big  $O(1)$  and lots of extra memory allocations
  - You can have 100's of resources (or more) involved in rendering a frame
  - And many 100's to 1000's of draws.. so overhead adds up

# But, 32 batches should be enough

- We anyways want to limit unflushed batches during game/level startup during texture uploads
- And it is enough for 2x mipmap gen for largest possible texture
  - Normal u\_blitter batches flushed immediately
    - so never come close to 32 upper limit
  - But needed transiently for back-blits
- This turns every hashset of batches into a 32b bitmask!

# Nice things about bitmasks..

- Hash set ops:
  - insert  $\rightarrow |= (1 \ll \text{batch-}\rightarrow\text{idx})$
  - test  $\rightarrow \& (1 \ll \text{batch-}\rightarrow\text{idx})$
  - remove  $\rightarrow \&= \sim(1 \ll \text{batch-}\rightarrow\text{idx})$
  - iterate  $\rightarrow$  loop of `ffs()` (ie, `u_bit_scan()`)
- When you have many 100's of draws per batch, and up to 16 textures / 32 vbo's / N ubo's / TF streamout bo's, quero bo's, etc, it is nice to keep the overhead down

# So basically..

- All hash sets go away except batch->resources
  - Tests for inclusion guarded by `& (1 << batch->idx)`
    - So only do hash set insert for resources that aren't already referenced by the batch
  - Probably could go away if we merged `libdrm_freedreno` and `gallium`
    - vc4 does something like this..
    - But that would mean throwing away `kgsl` and `a2xx` support
    - Probably worth doing eventually, but not yet

# Results

- supertuxcart: +30%
  - new render engine has mid-frame UBO updates
- manhattan: +20%
  - mid-frame texture upload + generate-mipmap
- glmark2
  - desktop: +7%
  - shadow: +20%